

In order to launch DAAL spark samples you need to run spark on yarn.

How to install and run Hadoop & yarn can be found [here](#).

How to launch spark on yarn can be found [here](#).

How to run

In order to use DAAL, you need to follow a few simple steps

Step 1: scala installation

First you need to install scala and properly configure the environment

Download and install scala and add the necessary jars to the CLASSPATH env variable.

```
1. # download and install scala
2. wget https://downloads.lightbend.com/scala/2.11.12/scala-2.11.12.rpm
3. sudo rpm -i scala-2.11.12.rpm
4. export SCALA_JARS=/usr/share/scala/lib/scala-library.jar
5.
6. # adding needed jars to CLASSPATH env variable
7. declare -a jars=(
8.     "spark-core_"
9.     "spark-sql_"
10.    "spark-catalyst_"
11.    "spark-mllib_"
12.    "hadoop-common-"
13.    "jackson-annotations-"
14.    "breeze_"
15.    "breeze-macros_"
16.    "hadoop-mapreduce-client-common-"
17.    "hadoop-mapreduce-client-core-"
18.    "spark-tags_"
19.    "spark-unsafe_"
20.    "log4j-"
21. )
22.
23. for i in "${jars[@]}"
24. do
25.     export CLASSPATH=`sudo find / -name "${i}*.jar" | head -n 1`:$CLASSPATH
26. done
```

If needed, correct scala version in this script (scala samples validated on scala 2.11.11).

Step 2: downloading and setting up daal environment

If DAAL is already installed and the environment is configured in correct way, then just download DAAL Spark samples.

```
1. # download daal from the main repository
2. git clone https://github.com/intel/daal.git
3. # go to the scala spark directory
4. cd daal/samples/scala/spark
5.
```

Download the latest DAAL release and customize your environment.

```
1. url="https://github.com/intel/daal/releases/download/2019_u4/l_daal_oss_p_2019.4.007.tgz"
   # insert link to last daal release here
2. name="l_daal_oss_p_2019.4.007"
3. # download and unzip daal release
4. wget $url
5. tar -xzf "${name}.tgz"
6. # download daal from the main repository
7. git clone https://github.com/intel/daal.git
8. # create a working directory
9. mkdir daal_release_lnx
10. mv $name/daal_prebuild/linux/* daal_release_lnx/
11. mv daal/samples daal_release_lnx/daal/
12.
13. cd daal_release_lnx/daal
14. # setting up the build environment
15. source bin/daalvars.sh intel64
16.
17. cd samples/scala/spark
18.
```

Step 3: configuring tbb threads

DAAL uses TBB to achieve high parallel performance. TBB is a widely used C++ library for shared memory parallel programming and heterogeneous computing (intra-node distributed memory programming).

In the case of spark within each spark executors, the machine learning algorithms are computed using TBB. Unfortunately, TBB threads in spark executors are currently not configurable. This means that you need to set them yourself. You can do this as follows

```
1. sudo find / -name spark-defaults.conf
```

if the cluster is started on aws emr, then the path should look something like this:

/etc/spark/conf.dist/spark-defaults.conf

Caution, you need sudo to edit the file.

And in the found file you need to add:

```
1. spark.task.cpus      8
```

Step 4: starting test samples

In order to verify that everything works, you can run tests, using launcher from attachment.

Firstly, remove default launcher:

```
1. rm launcher.sh
```

Instead of the removed launcher take the launcher from

https://raw.githubusercontent.com/intel/daal/2019_u4/samples/scala/spark/launcher.sh.

It corresponds to the DAAL prebuild, that used in this article.

Make the launcher executable.

```
1. chmod 744 launcher.sh
```

Start samples:

```
1. ./launcher.sh intel64
```

This will run the tests on small test data.

When running the tests, the warning `cat: /etc/alternatives/jre/release: No such file or directory` may be displayed. This warning is triggered by `scalac` and can be ignored. It does not affect the result.

So, if all the tests were successful, you can proceed to write a test case.

Consider launching DAAL on Spark using the example of PCA.

Step 1: adding test data

Add test data.

Firstly, make sure, that you are in `daal/samples/scala/spark` directory.

If the tests were run, then the test data should already be in hdfs.

If not, then you can put test data on hdfs with the following command.

```
1. export sample=PCA
2. hadoop fs -mkdir -p /Spark/${sample}/data
3. hadoop fs -put ./data/${sample}*.txt /Spark/${sample}/data/
```

Step 2: test file creating

Create a test file `PcaTest.scala`.

Lets import the necessary modules.

```
1. package DAAL
```

```

2.
3. import org.apache.spark.SparkConf
4. import org.apache.spark.SparkContext
5. import daal_for_mllib.{PCA, PCAModel}
6.
7. import org.apache.spark.mllib.linalg.Vectors
8. import org.apache.spark.mllib.linalg.Matrix
9. import org.apache.spark.mllib.linalg.distributed.RowMatrix
10.
11. import java.io._

```

In the PcaTest.scala file, add a PCA call.

```

1. object PcaTest extends App {
2.   val conf = new SparkConf().setAppName("Spark PCA")
3.   val sc = new SparkContext(conf)
4.
5.   // Retrieve input data from TXT and load it into RDD
6.   val data = sc.textFile("/Spark/PCA/data/PCA.txt")
7.   val dataRDD = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble))).cache()
8.
9.   // Compute PCA decomposition with 10 principal components
10.  val model = new PCA(10).fit(dataRDD)
11.
12.  // Print resulting eigenvectors, eigenvalues and the explained variance
13.  println("Eigen vectors: " + model.pc.toString())
14.  println("")
15.  println("Eigen values: " + model.explainedVariance.toString())
16.  sc.stop()
17. }

```

Step 3: setting up environment

Set up the environment.

```

1. export daal_ia=intel64
2. export CLASSPATH=${SCALA_JARS}:$CLASSPATH
3. export SHAREDLIBS=${DAALROOT}/lib/${daal_ia}_lin/libJavaAPI.so,${DAALROOT}/../tbb/lib/${daal_ia}_lin/gcc4.4/libtbb.so.2,${DAALROOT}/../tbb/lib/${daal_ia}_lin/gcc4.4/libtbbmal
  loc.so.2

```

Step 4: building test project

Build a project.

```

1. scalac -d PcaTest.jar PcaTest.scala <path-to-daal-spark-samples>/sources/PCA.scala

```

When running the tests, the warning `cat: /etc/alternatives/jre/release: No such file or directory` may be displayed. This warning is triggered by `scalac` and can be ignored. It does not affect the result.

Step 5: launching test project

Launch the project.

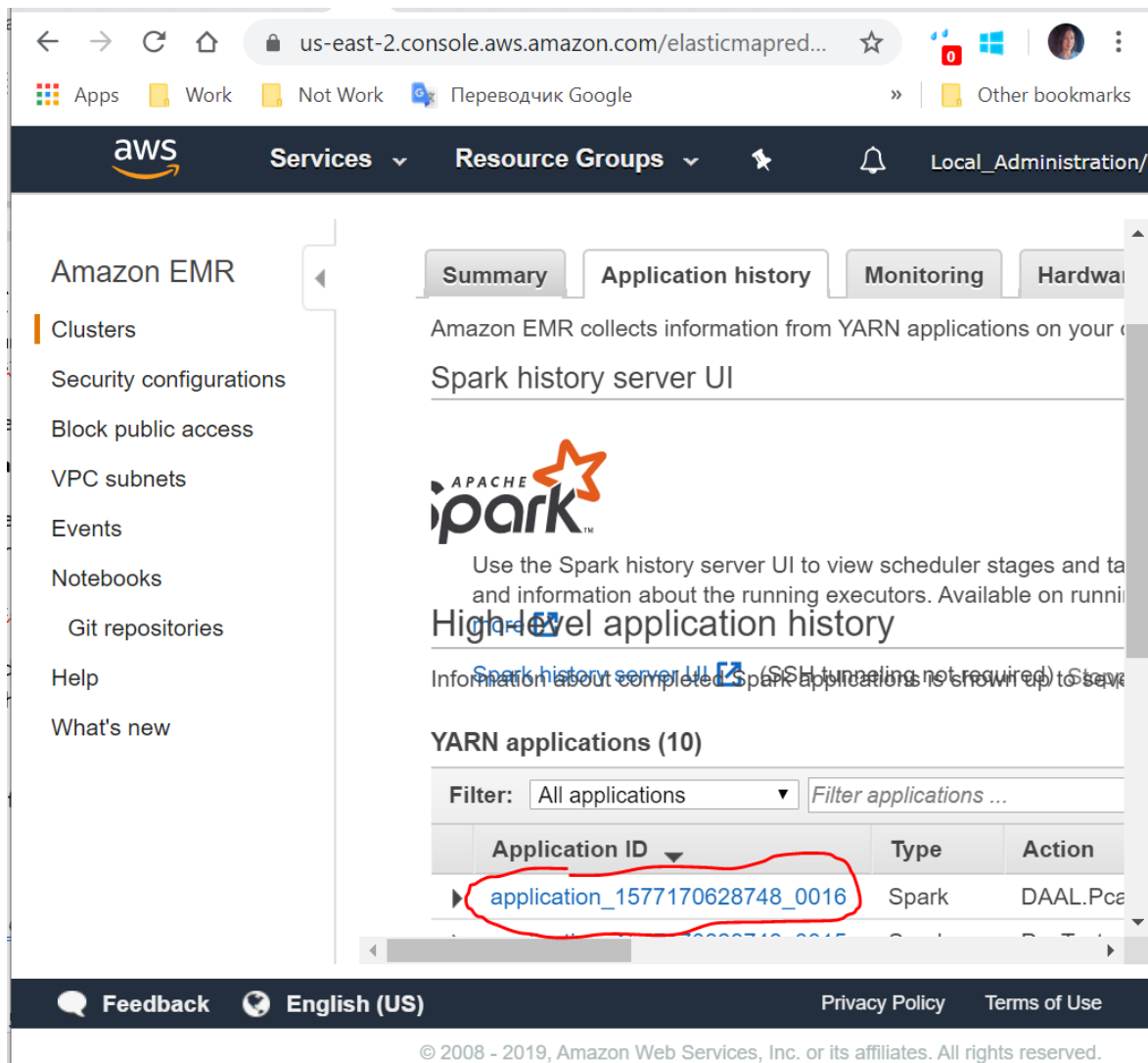
```
1. spark-submit --driver-class-path \"${DAALROOT}/lib/daal.jar:${SCALA_JARS}\" --  
jars ${DAALROOT}/lib/daal.jar --files ${SHAREDLIBS} -v --master yarn-cluster --deploy-  
mode cluster --num-executors 12 --executor-cores 8 --executor-memory 15G --  
class DAAL.PcaTest PcaTest.jar
```

Description of the num-executors and executor-cores parameters and their correct settings described later in **How to configure a spark in correct way** chapter.

If everything is ok, then the message: final **status: SUCCEEDED** will be displayed in the spark logs, and you can get eigen values and vectors using the command:

```
1. yarn logs -applicationId <application_id> > pca_logs.txt
```

Application id can be found from the list of applications, which can be obtained in Spark UI or application history tab, if the cluster was deployed on AWS EMR.



The screenshot shows the AWS Management Console interface for Amazon EMR. The left sidebar contains navigation links for Amazon EMR, Clusters, Security configurations, Block public access, VPC subnets, Events, Notebooks, Git repositories, Help, and What's new. The main content area is titled 'Amazon EMR' and includes tabs for Summary, Application history, Monitoring, and Hardware. The 'Application history' tab is selected, showing a list of YARN applications. The table has columns for Application ID, Type, and Action. The application 'application_1577170628748_0016' is highlighted with a red circle. The application is of type 'Spark' and has a status of 'SUCCEEDED'.

Application ID	Type	Action
application_1577170628748_0016	Spark	DAAL.Pca

It will create `pca_logs` text file, where eigen vectors and eigen values can be founded. Logs will look like this:

Eigen vectors:

```
0.25798164372921656  -0.4292620757958274      ... (10 total)
```

Eigen values:

```
[0.3779671568979385, 0.1750187552716385, 0.12002756104610995, ...
```

How to configure a spark in correct way

Num of executors and executors cores

number of executors and cores per executor can be set with next conditions:

All cores of cluster \geq (number of executors) \times (cores per executor)

(Number of executors) / (Number of nodes) is integer (all nodes will have same amount of executors)

For example, if you have 5 nodes with 72 cores per node, you might set 20 executors with 18 cores per executor (it's mostly better when number of nodes < number of executors).

In my case I have 6 nodes with 16 cores per node.

Memory per executor

In general the memory per executor can be calculated by the formula:

```
Total_ram_memory_per_node/num_executors
```

However small overhead memory is also needed to determine the full memory request to YARN for each executor.

The formula for that overhead is $0.07 * \text{spark.executor.memory}$

Thus, the memory per executor can be calculated by the formula

```
Total_memory/num_executors - (Total ram per node  
memory/num_executors*0.07)
```

Using Different Garbage Collector

To improve performance, you can use different garbage collector. In the case of daal, G1 GC gave the best performance.

As experiments show, better performance can be achieved using the following g1 garbage collector settings:

- XX:MaxGCPauseMillis=50
- XX:ParallelGCThreads=node_cores

To use it, add the argument conf:

```
1. spark-submit --driver-class-path "${DAALROOT}/lib/daal.jar:${SCALA_JARS}" --  
jars ${DAALROOT}/lib/daal.jar --files ${SHAREDLIBS} -v --  
conf "spark.executor.extraJavaOptions=-XX:+UseG1GC -XX:MaxGCPauseMillis=50 -  
XX:ParallelGCThreads=16" --master yarn-cluster --deploy-mode cluster --num-  
executors 12 --executor-cores 8 --executor-memory 15G --  
class DAAL.PcaTest PcaTest.jar
```